UCRL_JC_155202

Computational Challenges in Nuclear Weapons Simulation

Charles F. McMillan, Thomas F. Adams, Michel G. McCoy, Randy B. Christensen, Brian S. Pudliner, Michael R. Zika, Patrick S. Brantley, Jeffrey S. Vetter, John M. May

This paper was submitted by Lawrence Livermore National Laboratory for presentation at the “Future of Supercomputing” workshop sponsored by the National Research Council’s Computer Science and Telecommunications Board in Santa Fe, MN, September 24-25, 2003.

August 29, 2003

This work was performed under the auspices of the U.S. Department of Energy by the University of California, Lawrence Livermore National Laboratory under Contract No. W-7405-Eng-48.

Lawrence Livermore National Laboratory
P.O. Box 808, Livermore CA 94551

UCRL_JC_155202

Computational Challenges in Nuclear Weapons Simulation

Charles F. McMillan, Thomas F. Adams, Michel G. McCoy, Randy B. Christensen, Brian S. Pudliner, Michael R. Zika, Patrick S. Brantley, Jeffrey S. Vetter, John M. May

Introduction

After a decade of experience, the Stockpile Stewardship Program continues to ensure the safety, security and reliability of the nation's nuclear weapons. The Advanced Simulation and Computing (ASCI) program was established to provide leading edge, high-end simulation capabilities needed to meet the program's assessment and certification requirements. The great challenge of this program lies in developing the tools and resources necessary for the complex, highly coupled, multi-physics calculations required to simulate nuclear weapons.

This paper describes the hardware and software environment we have applied to fulfill our nuclear weapons responsibilities. It also presents the characteristics of our algorithms and codes, especially as they relate to supercomputing resource capabilities and requirements. It then addresses impediments to the development and application of nuclear weapon simulation software and hardware and concludes with a summary of observations and recommendations on an approach for working with industry and government agencies to address these impediments.

History

The nuclear weapons program has relied on state-of-the-art supercomputers since its inception. Even with simplified physics models and under-resolved numerics, weapons simulations have provided insight and information unobtainable in any other way. Consequently, the nuclear weapons laboratories have always been at the forefront of supercomputer development, working hand-in-hand with the nation's foremost computer manufacturers to develop the world's fastest computers.

From the late 1970s through the early 1990s, Cray vector supercomputers provided the bulk of the computational capabilities at the weapons laboratories for two reasons: (1) the serial nature of our codes placed a premium on single processor performance; and (2) the Cray line of vector supercomputers provided a combination of world-leading scalar speed, short vector capability, and cost performance that made them a natural choice for the weapons laboratories. They were both fast and cost-effective across the full spectrum of our weapon simulation codes.

In order to take full advantage of the Crays, the laboratories made major investments in both application and system software. Laboratory teams completely rewrote major simulation codes to maximize the fraction of work that could be vectorized, wrote vectorizing compliers, and developed the first time-sharing operating system for Cray computers. Much of the world's expertise in these areas was developed at the weapons laboratories.

It is important to note that, even after all this effort, the large, multi-physics applications that dominate our workload still displayed a relatively large scalar fraction (~20%), since the algorithms that provided the shortest time to solution were often not the ones most amenable to vectorization. The vector capabilities of the Crays provided a speed advantage of three to four over scalar speed for these codes. Long vector machines, or vector machines with relatively slow scalar speeds, were not well suited to these applications.

In the early nineties, the convergence of three events led to the end of the Cray supercomputer era at the laboratories and the beginning of the move to parallel computing.

1. Inexpensive commodity “killer micros” emerged with scalar speeds equaling or exceeding those of the custom processors used in the vector supercomputers.
2. Parallel computing technology reached the point where it could begin to support the development and use of large, multi-physics simulation codes.
    - The speed of the individual processor was no longer of paramount importance.
3. The decision to halt underground nuclear testing generated a requirement for a much higher fidelity weapon simulation capability, one that could not be obtained through any realistic increase in single-processor performance.
    - Simply reducing numerical errors to the point where they no longer mask inadequacies in physical models requires ~100 teraFLOPS. Incorporating improved physics models into the codes and applying them on a production basis to stockpile issues easily takes us to the petaFLOPS regime.

These events led to the creation of the ASCI program to develop the applications, physics models, computers, and infrastructure to enable terascale simulation via massive parallelism. Our current path to 100 teraFLOPS and beyond depends on accelerating the development of the hardware and software needed to build tightly integrated, massively parallel supercomputers using commodity components. Given the enormous expense in developing a next-generation computer, it is crucial that we fully leverage the investments made by processor and computer vendors for the larger marketplace.

ASCI Platforms and Programming Environment

Platforms

Over the last five years, the ASCI Program has acquired a variety of increasingly powerful computing platforms. As Table 1 shows, these systems have come from a number of vendors and have featured a variety of processors. The portability of applications among machines has been relatively smooth because of the general commitment to standard languages and programming models (e.g., MPI parallelism with some use of multithreading) and avoidance of processor-specific optimizations. As a result, the ASCI community has been able to take advantage of rapidly evolving processor technology as it has become available.

By outpacing the computer industry’s historic annual rate of growth in computing power, the ASCI platforms have been growing in size and power requirements, which has led to significant investments in facilities and power. As the last line of the table shows, this trend is not unique to commodity-based processors: the Earth Simulator also required a significant investment in facilities and power. Goals of technology exploration projects, such as the ASCI Linux Cluster and the Blue Gene/L project, have been to reverse this trend. Systems like these are expected to be notably smaller and consume less power than the current generation of supercomputers. The Blue Gene/L project, in particular, takes advantage of power- and space-saving technology developed for the embedded processor market.

Table 1. Deployed and planned ASCI Computer Systems

System	Host	Installation Date	Vendor	Number of Processors	Peak Floating Point Rate (TFLOPs)	Processor Type	Footprint (ft^2)	Power (MW)	Construction Required
Red (upgrade)	SNL	1997	Intel	9,298	3.1	Pentium II Xeon	2,500		Use existing
Blue Pacific	LLNL	1998	IBM	5,808	3.89	PowerPC 604e	5,100	0.6	Use existing
Blue Mountain	LANL	1998	SGI	6,144	3.072	MIPS R10000	12,000	1.6	Use existing
White	LLNL	2000	IBM	8,192	12.3	POWER3-II	10,000	1.0	Expand existing
Q	LANL	2002	HP/Compaq	8,192	20.5	Alpha EV 68	14,000	1.9	New
ASCI Linux Cluster	LLNL	2003	IBM	1920	9.2	Pentium 4	1,200	0.48	Use existing
Red Storm*	SNL	2004	Cray	10,368	40	AMD Opteron	3,000	2.1	New
Purple*	LLNL	2004	IBM	12,220	100	POWER5	12,000	4.5	New
BlueGene/L*	LLNL	2004	IBM	131,000	180/360	PowerPC 440	2,500	1.8	Use existing
Earth Simulator	ES	2002	NEC	5,120	40	NEC SX-5 descendant	34,000	10	New

* Planned systems

Programming Environment

ASCI programming environments stress most software development tools due to the scale of the hardware architecture, software complexity, and the requirement of compatibility/ portability across the ASCI platforms. Development, testing, and validation of multi-physics application codes require four to six years. The productive lifespan of these codes is at least ten years and may span several decades. Therefore, these applications must span not only today's computers, but also possible future systems.

With the substantial ASCI investment in applications and libraries, the choice of programming model takes on strategic importance. A high degree of code portability and longevity is a major objective. ASCI codes must execute at all three ASCI sites located at Lawrence Livermore National Laboratory (LLNL), Sandia National Laboratories, and Los Alamos National Laboratory. Codes are developed in standards-conforming languages; so non-standard compiler features are of little interest unless they can be made transparent. We also do not take advantage of idiosyncratic features of optimization, unless they can be hidden from the codes (e.g., in a standard library); processor-specific optimizations are discouraged.

As Fig. 1 illustrates, for almost 25 years, our community could rely on a relatively stable development environment. For example, users programming on the CDC 6600 in 1972 could rely on the same vendor and general operating environment for nearly 15 years. In the mid-1990s, this operating mode changed, so that users had access to a range of platforms with little guarantee of longevity for any particular platform or software environment. In contrast to the earlier period, this wave of diversity brought new attention to the portability and compatibility of programming models, languages, and software environments.

As a consequence of ASCI's platform diversity, virtually all applications at LLNL today use a combination of three programming models: the serial model, the symmetric multiprocessor model using OpenMP, and the message-passing model using the message passing interface (MPI). Most applications use the MPI for coarse-grain concurrency, and OpenMP, or possibly POSIX threads, for fine-grain concurrency. Therefore, it is crucial that any platform include a scalable, high-performance MPI environment with low-latency.

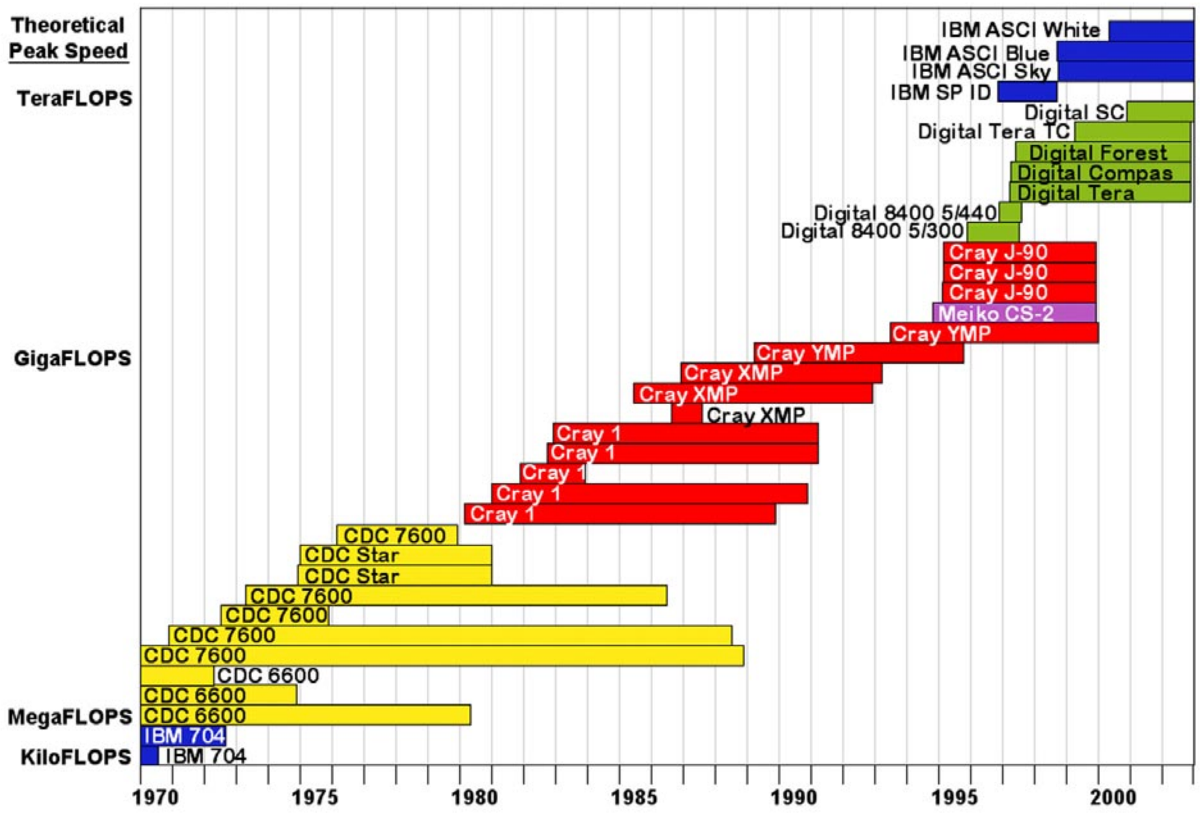

Figure 1. LLNL computer architecture and programming model history.

All three of these programming models, when combined with standard languages, such as ANSI C, C++, and Fortran90, provide a portable and compatible software environment for applications development across all major platforms currently available on the market. ASCI applications rely on a range of languages for software development, often using mixed languages to develop one application. ASCI uses a variety of compilers and numerous libraries provided by the platform vendors, independent software vendors, or by other scientific institutions. ASCI has also invested in software tools to provide a consistent development environment across important platforms. With this approach, users can migrate among platforms without learning different tools.

Algorithm and Code Development

Teams of approximately 10 to 25 individuals develop major ASCI codes. These teams include both "code physicists" and computer scientists. Code physicists typically have extensive training (Ph.D. or equivalent) in physics or a related discipline as well as practical knowledge of computer science. They work with the end users of the codes to understand their needs and develop efficient algorithms to model various physical phenomena. The computer scientists on the team do code design, development, and maintenance, as well as performance analysis and related activities. They typically have a working knowledge of the physics being modeled in the software. A typical large code might have half a million or more lines of source code.

Algorithm development balances the (often competing) requirements of high-fidelity physics, short run time, parallel scalability, and algorithmic scalability (i.e., constant rate of convergence as number of unknowns increases). These requirements have eliminated some algorithms; for example, parallelizable multigrid algorithms have generally displaced inherently serial Gauss-Seidel iterative techniques. Thus, algorithm development has, in some ways, been influenced by target architectures.

Because the number of users of these codes is relatively small (a few dozen people, who often have close contact with the code teams), and because their needs evolve rapidly, the codes are under continuous development. Despite the fact that a code (or parts of it) may remain in use for decades, it is not unusual for new releases to be made weekly. Different users and different problems exercise various capabilities of a code, so they are typically written in a framework that allows users to choose among packages and algorithms.

Characteristics of Codes and Algorithms

Approach to Code Development

Typical simulations are composed of multiple physics packages, which advance a shared set of data throughout the problem simulation time. While the details vary among packages, all implementations require that multiple physics packages run concurrently. These packages often have competing computation and communication requirements. Generally, our strategy is to compromise among the various competing needs of these packages.

Our codes not only contain many different physics packages, but are also called upon to calculate a great variety of problems. The mix of problems requires that all packages perform efficiently. Furthermore, no one loop, algorithm, or package is sufficient to characterize the performance of the entire code.

The overriding principle governing the development of our codes is to attain the maximum degree of accuracy in the minimum amount of time. This principle tends to favor intelligent algorithms that sacrifice computational efficiency (e.g., in terms of percentage of peak FLOPS) for quick turn-around times. For example, much faster solutions for certain problems may be attained using adaptive mesh refinement (AMR) methods, at the cost of much reduced per-zone computational efficiency.

Since the characteristic length scale for each type of physics in a multi-physics calculation can vary greatly, there is no single fixed resolution requirement for a calculation. When faced with widely varying resolution requirements between a set of physics packages, we may either run the entire problem at the finest resolution or introduce more complex meshing strategies. The first solution can be prohibitive when we are forced to run computationally or memory-intensive packages at finer-than-required resolution. The second solution carries a computational and accuracy cost associated with coordinating physics carried out on different physical scales, so the optimal solution is a compromise that is highly problem- and algorithm-dependent.

Due to the scale of the problems being solved (1-40 TeraByte peak in-core memory required) our applications employ distributed-memory parallelism. All but the least memory-intensive modules employ spatial domain decomposition in their distributed-memory parallel strategy. This approach allows us to solve simulations of unprecedented size; however, it requires various communication operations (point-to-point, collective, etc.) to properly synchronize the solution variables.

Shared-memory parallel (SMP) architectures have offered the opportunity to combine shared memory with distributed-memory parallelism. Shared-memory parallelism is typically obtained in our codes using Open MP threading directives. By taking advantage of threading we may reduce memory overhead by sharing large data structures among processors with no communication overhead. Thus in some cases, threading allows us to run much larger problems than we could with pure distributed-memory parallelism. Threads, however, have many disadvantages. Ensuring thread-safe execution and obtaining maximum threading efficiency are non-trivial tasks. On the other hand, since threading reduces the effective number of processors participating in a collective communication, we can take advantage of threads to improve the scaling of algorithms limited by collective communications.

Data and Memory Characteristics of Algorithms and Codes

Since our current platforms are exclusively cache-based architectures, we have worked very hard to ensure spatial and temporal locality in memory accesses. Spatial locality means that when one memory location is accessed, there is a high probability that neighboring memory locations will be accessed in subsequent calculations. Temporal locality is achieved when data from a single memory access is used repeatedly. By the nature of the problems we are trying to solve, we are, on occasion, driven to use algorithms that do not have ideal memory-access patterns. The access pattern descriptions in Table 2 give a representative sample of the packages contained in our codes. When there is a performance advantage to be gained, we will re-arrange the memory layout of the data shared between physics packages. In some cases (e.g., Packages A and D), the memory access is predictable and, therefore, it is relatively easy to arrange our data such that we obtain the best locality. The memory-access patterns of other packages (e.g., Package C) are extremely difficult to optimize since the algorithms themselves have a low degree of locality.

Table 2: Physics package memory-access characteristics.

Physics Package	Memory / Zone (KB)	Access Pattern
A	0.2	Predictable with a modest amount of spatial and temporal locality
B	50-80	Predictable, but difficult to optimize, low spatial but high temporal locality
C	0.5-100	Unpredictable memory access, low spatial and low temporal locality
D	0.5	Predictable, with medium to high spatial and temporal locality.

Like memory-access patterns, the memory-size requirements of the various packages differs considerably and are yet another source of compromise when determining the optimal decomposition for a problem. In some cases, we may run several packages at a sub-optimal decomposition in order to allow the most memory-intensive packages to fit into the memory of the machine. In Table 2, we can see the memory requirement per computational element for our representative set of packages. It is worth pointing out that the memory requirement per zone can fluctuate dramatically during the course of a calculation (e.g., Package C).

In addition to the per-zone cost for each package, there is also a fixed overhead per processor determined by the size of the infrastructure (data tables, mesh description, code image, etc.) needed by that package. For some classes of problems, this memory overhead can exceed 256 MB per MPI task. This overhead can be mitigated by taking advantage of shared memory parallelism or by compromising speed for memory footprint (e.g., computing some of the table elements on the fly rather than pre-computing them and caching the results).

Communication Characteristics of Algorithms and Codes

Table 3 shows a breakdown of the communications characteristics for an illustrative set of physics packages. The number of available degrees of parallelism can vary from a few to many, depending on the algorithms involved. Virtually all packages make use of some spatial decomposition of the problem. Some algorithms allow for parallelization in other parameter spaces. Given the diversity of communication requirements across these algorithms, it is unlikely that a single spatial decomposition will be optimal for the entire problem.

Table 3: Physics package communication characteristics.

Physics Package	Available Degrees of Parallelism	Type of Communications	Frequency of Communications	Message Size	Scaling Characteristics
A	Very Few	Point-to-point, exclusively surface communication	Regular, approx. 10 per computational cycle	60-200 KByte	Ratio of communicating surface area to computational volume
B	Many	Point-to-point, potentially some volume communication	Solution driven, but bounded	500 Kbyte to 2 MByte	Complex
C	Few	Point-to-point, potentially some volume communication	Random and asynchronous	Few bytes to 500 KByte	Complex
D	Very Few	Collective communications and point-to-point	Solution driven, tightly synchronized	Few bytes to 60 KByte	As collective communication scales

Our algorithms make extensive use of point-to-point communications, implemented using non-blocking, send-and-receive MPI calls. In some cases, the algorithms are highly dependent on collective communications, such as in the case of Package D, which is characterized by a large number of collective communications in a tight loop. Some algorithms require periodic mesh-sized communication that can stress the bandwidth capabilities of the network. Communication patterns can vary from regular and predictable (e.g., Package A) to essentially random and unpredictable (e.g., Package C). Since 5 to 10 such disparate physics packages may be integrated in a single calculation, it is rather difficult to point to a single defining communications characteristic.

Computational Characteristics of Algorithms and Codes

Codes are often characterized by the percentage of machine theoretical peak performance that they attain. Recent studies of our codes on the ASCI White machine show that they are getting 1-12% of peak. A recent survey of codes of similar complexity being developed at universities for non-weapons applications shows that those codes are achieving similar percentages of peak (i.e., 0.5-15%). For example, a very efficient code being developed at the University of Illinois to simulate the performance of solid rocket boosters is reported to give 7.7 - 14.5% of peak. Much higher percentages of peak speed could only be reached if all code instructions consisted of floating-point multiply-adds (FMAs) on data already residing in cache. Due to the compromises that are necessary in large-scale multi-physics simulations, codes of this kind, used in both weapons and non-weapons applications, rarely produce this instruction mix. Thus, this single aggregate measure is not sufficient to characterize code performance. Other metrics, as described below, provide more meaningful measures of the performance of complex multi-physics codes.

When attempting to characterize the computational nature of our algorithms, there are several things to consider. How much computational work is there per computational element? How much computational work is possible per memory access? How much computational work can be accomplished between communications? How is the computational work distributed spatially and temporally over the course of the calculation? Table 4 shows the tentative answers to these questions for some of our representative packages.

Table 4: Physics package computational characteristics.

Physics Package	Millions of Floating-Point Operations per Zone per Cycle	Computational Intensity	Ratio of Computation to Communication	Load Balance Characteristics
A	0.01 – 0.05	0.5 – 1.0	Moderate ratio of computation to communication	Can have large spatial and temporal imbalance
B	2 - 4	> 1.5	Very large ratio of computation to communication	Challenge to spatially load balance
C	0.01 – 0.05	0.1 - 0.2	Varies from very low to very high	Dominated by spatial and temporal load imbalance
D	0.08 - 2	0.8 – 1.0	At points, very small amount of computation to communication	Very load balanced

We are measuring the computational characteristics of our algorithms as the necessary tools become available. Since the bulk of our packages rely on floating-point operations, the amount of floating-point work per zone is a measure of the computational work. “Computational intensity,” which is the ratio of floating-point operations to memory operations, gives a measure of how efficiently we make use of the data brought in from memory for our floating-point work. For cache-based, super-scalar architectures, a high computational intensity generally translates into high performance. We can see from Table 4 that our packages represent a broad range of floating-point work in these metrics. Package C stands out for its particularly low computational intensity because its instruction mix is weighted more toward fixed point than floating-point operations. We are reminded that a single metric for performance is unlikely to be suitable for all packages. As we use performance tools to analyze our algorithms on a variety of platforms, we are working to develop a collection of metrics to represent overall processor efficiency.

Like computational intensity, the amount of work that can be accomplished between communications is a measure of how efficiently we can make use of the data local to the processor. As yet, we have not quantified this metric; but Table 4 gives a qualitative description by package. Not only do we have a range of ratios between packages, some packages (e.g., Package C) can even vary greatly between problems or even within a single run.

Load balance is a measure of how evenly the work is distributed across processors. For problems that are decomposed spatially, due to the physics involved, the computational work can vary in different spatial parts of the problem. In this case, an optimal decomposition is usually sought at the outset of the problem and fixed over the course of the calculation (e.g., Package D). Frequently, in our calculations, the spatial load balance evolves quite radically over the simulation time (e.g., Packages A-C). In such cases of time-varying load imbalance, we seek to employ some dynamic load-balancing scheme. The degree to which this can be done cost-effectively is driven by the nature of the parallelism in the problem and how efficiently the work may be redistributed. It is safe to say that we are still in our infancy in understanding how to efficiently load balance over large numbers of processors in multi-physics codes.

Some Impediments to Achieving Programmatic Goals

In this section, we identify a number of issues that have challenged the ASCI Program to date, and some of the approaches the program has adopted to address these issues. These issues are general in nature, and are not unique to ASCI. In succeeding sections, we offer some recommendations, based on our experience, suggesting how the broader community and the government might work together to address these impediments in a sustainable and efficient manner.

Capability and Capacity

Although increases in computing power have allowed ASCI codes to improve vastly the quality of physics simulations, these have only begun to reach the level of fidelity required for the ASCI Program to carry out its mission. As was confirmed recently by the JASONS[1], further increases in computing power (i.e., capability) will clearly be necessary over the next decade to reach peak computational capability levels of a petaFLOPS and beyond. Yet, the path to such capability is not fully understood and requires research, development, and testing of "capability exploration" machines.

Further, capability cannot stand alone. As was also emphasized by the JASONs, weapons studies (like most scientific studies) invariably require a spectrum of capacity and capability calculations. This argues for an appropriate mix of capacity and capability resources. Our current estimates suggest a ratio of approximately one to one.

Within the ASCI Program recently, there has been a concerted effort to exploit low-cost clusters to address capacity computing requirements. LINUX clusters with high-performance switches and Open Source software are now running, fully automated through batch queues, with multiple users running up to 2,000 processors. Such capacity choices can indeed reduce cost, and as the full complement of Open Source software (like file systems, schedulers, checkpoint restart) and scaleable tools become available, costs will decrease further.

ASCI has participated in this arena by encouraging the evolution of low-cost vendor agnostic clusters with Open Source Software through multiple PathForward contracts in the extensible file systems and tools areas.

The Memory Wall

A discussion earlier in this paper identified the complex data-management problems that limit the speed of memory access. These problems arise because the great increase in processor speed in recent years has not been matched by comparable increases in the speed of communication to memory or between processors. This disparity in speeds creates a bottleneck often referred to as a "memory wall." The problem manifests itself on the node itself and in communications between nodes. Experience has shown that it has been exceedingly difficult for the program to affect the design of processors or nodes. Design begins years ahead of commercial delivery and these components are designed to meet the needs of the huge commercial market not the modest high-performance computing market. Improving the performance of switches and associated adapters has been almost as challenging.

Cost

Although the ASCI program has aggressively pushed the limits of high-end computing, it has had to do so within the constraints of increasingly tight budgets. High levels of performance are essential, but so are reasonable costs. An important element of the ASCI strategy has been to exploit the rapidly declining costs of commodity computing components. An approach here is to encourage the vendor to combine these commodity components in a scalable and effective manner. For instance, the BlueGene/L computational research system at 64,000 nodes, uses low-power, low-cost embedded processors with floating-point units, system-on-a-chip technology, and fully three interconnects. None of this is revolutionary except possibly for how all these commodity pieces are combined. Another way the program has controlled costs is by adhering to a common programming model, so that codes can be ported to new machines without incurring large expenses (and delays) for rewriting codes. A third method has been mentioned above: relying on a combination of vendor-integrated systems and clusters to achieve the appropriate, lowest cost mix of systems on the floor.

Programmability

The MPI model of parallelism has been effective for developing portable codes, but many people have observed that its low level of abstraction makes programming more difficult than necessary. Also, as discussed previously in this paper, the single program, multiple datastream (SPMD) paradigm implicit in MPI can lead to difficult load balance problems, especially for massively parallel applications. Numerous alternative parallel programming models have been proposed. However, no model currently offers the combination of portability, robustness, availability, and efficiency that MPI does. We believe that further improvements in program development efficiency will require a new programming model that will replace MPI, but any new model must have broad community support and offer a straightforward upgrade path from MPI codes. Most importantly, it must offer portability across a range of systems, currently and over the next 10-15 years.

Stability

The *annual* ASCI investment in applications and supporting science is of the same order as the platform and computing environments budget. However, the applications investment is *cumulative* since an application lives for over fifteen years whereas computers live only a few years. Therefore, the current existing investment in applications is on the order of $1-2B for ASCI alone, whereas the cost of a large computer is on the order of $100M. This highlights the importance of stability and continuity in the programming environment. New computing environments or architectures that might appear in the future and that, by their nature, require massive rewriting of codes, risk being ignored unless the promise is extreme or the transition, through software, is transparent.

Summary

- Tightly coupled, multi-physics nuclear weapons codes cannot be characterized by simplistic parallelization or optimization strategies.
- ASCI has adopted effective programming models, which continue to provide a portable and flexible software development environment.
- Close coupling with users has enabled responsive development of codes with increasing capability to meet programmatic needs.
- Our weapon simulation codes exhibit performance characteristics comparable to other complex multi-physics codes used for non-weapons applications.
- Partnerships and collaborations with the computer industry have enabled effective machine acquisitions and technology development and will be the key to meeting future requirements.

Recommendations

Given the formidable problems faced by the high performance computing community, the deliberate and reasoned approach adopted by the CSTB Study Committee in its interim report is commendable. In particular, we resonate strongly with the views expressed regarding the:

- Lack of scalable software and the existence of problems related to the memory wall,
- Need for sustained investment in both evolutionary and innovative technologies, and
- Need for a role for government even though it "appears that the ability of government to affect the supercomputing industry had diminished because supercomputing is a smaller fraction of the total computer market and computer technology is increasingly a commodity".[2]

We see that the fundamental challenge facing this Study Group is to define the precise role of government in leveraging a world market. We recommend government investments in both the evolutionary and innovative arenas.

Evolutionary Investment

There is justification for substantial investment in scaleable tools, primarily residing in the Open Source community, but not excluding third-party vendors, when appropriate. This raises the thorny problem of standards, but standards might evolve under the aegis of a multiple government agency forum, especially if development dollars are available.

Government investments could affect processor, node, and interconnect performance, if made early enough in the vendor requirements and design process. Successful investments of this kind could speed the availability of very low cost, commodity, high-performance, 64-bit clusters that scale to the 100-1000 teraFLOPS level this decade.

Government should also invest in creative combinations of commodity components, as was indicated earlier in the discussion of the IBM BlueGene/L.

Innovative Research

It is important for the government to fund innovative research to avoid a narrow focus on incremental improvements at the expense of revolutionary improvements. Evolutionary improvements, as mentioned by the JASONS, may not be a clear path to petaFLOPS. Even the most brilliant innovation must sustain itself in the market or the vendor community will ignore it. The alternative to market sustainability is an enterprise that lives on the margins whose survival depends on governmental support not only for research and development but also for manufacture. A long-term winning strategy must target innovation that will earn broad market acceptance.

Appendix

These URLs contain summaries of the existing generation of ASCI (and other DOE) platforms:

http://www.llnl.gov/asci/platforms/
http://www.sandia.gov/ASCI/Red/
http://www.llnl.gov/asci/platforms/bluepac/
http://www.llnl.gov/asci/platforms/white/
http://www.lanl.gov/asci/bluemtn/
http://www.llnl.gov/linux
http://www.lnxi.com/news/lightning_info.php

This work was performed under the auspices of the U.S. Department of Energy by the University of California, Lawrence Livermore National Laboratory under Contract No. W-7405-Eng-48.

[1] Roy Schwitters, *Requirements for ASCI, Executive Summary*, JASON Program Office, July 29, 2003.
[2] *The Future of Supercomputing, An Interim Report*, National Research Council, August 12, 2003.